Incompressible Navier-Stokes Baseline Performance Measurement

P. Colella
D. F. Martin
N. D. Keen

Applied Numerical Algorithms Group
NERSC Division
Lawrence Berkeley National Laboratory
Berkeley, CA

October 17, 2002

Benchmark Problem Description

The algorithm is described in a separate document entitled "Incompressible Navier-Stokes Software Design Document".

To evaluate the performance of the incompressible Navier-Stokes AMR code, we use two co-rotating vortex rings in three dimensions. For this problem, the vorticity distribution is specified, and the initial velocity is then computed based on the initial vorticity field. Each vortex ring is specified by a location of the center of the vortex ring (x_0, y_0, z_0), the radius of the center of the local cross-section of the ring from the center of the vortex ring r, and the strength of the vortex ring Γ.

The cross-sectional vorticity distribution in each vortex ring is given by

$$\omega(\rho) = \frac{\Gamma}{a\sigma^2} e^{(\frac{\rho}{\sigma})^3} \tag{1}$$

where ρ is the local distance from the center of the ring cross-section, $a = 2268.85$, and $\sigma = 0.0275$.

For this problem, the first vortex ring is centered at $(0.5, 0.5, 0.4)$, with a radius of 0.2, and strength Γ of 1.5. The second vortex ring is centered at $(0.5, 0.5, 0.65)$, with a radius of 0.25 and a strength $\Gamma = 1.0$.

This document presents serial profiling and parallel performance results for a given input to the incompressible Navier-Stokes AMR code. The physical domain size of the primary problem investigated in the work is 32x32x48. There are 3 levels of refinement in the AMR grid hierarchy, with a factor of four refinement between each pair of levels. The refinement tagging is based on the vorticity magnitude

The inputs file for the $32 \times 32 \times 48$ benchmark run is shown in Figure 1.

Target Platform and Compilers

The target platform for this benchmark measurement is a machine named named Halem located at GSFC. Halem is the NCCS Compaq AlphaServer SC45 System and it currently consists of 416 user-available processors. The halem processors are clustered into 104 symmetric multiprocessor nodes (4 processors per node). An additional set of 12 nodes is allocated as system and spare nodes. Halem is a hybrid system in the sense that memory is distributed among nodes, but within a node memory is shared.

The Fortran compiler used for this was the native Fortran compiler `f77` with the `-fast` optimization flag. The C++ compiler used was the GNU g++ compiler (version 3.1) with flags as `-O2 -ftemplate-depth-27 -Wno-long-long`.

Profiling Methodology

The primary metric used in this performance analysis work is wall-clock time of various units and sections of the code. The standard C function `gettimeofday()` is used to

```
# size=32x32x48 factor=50 timesteps=4 regrid_interval=4
main.max_step = 4
main.max_time = 200.0

main.num_cells = 32 32 48
main.max_level       = 2
main.ref_ratio       = 4 4 4
main.regrid_interval  = 4 4
main.block_factor    = 8
main.max_grid_size   = 32
main.fill_ratio      = 0.8
main.grid_buffer_size = 1
main.is_periodic     = 0 0 1
main.cfl             = 0.5

main.checkpoint_interval = -1
main.plot_interval       = 0
main.plotPrefix          = pltNew.
main.verbosity           = 2        # higher number means more verbose

ns.vorticity_tagging_factor = 0.0050

ns.init_shrink   = 1.0
ns.tag_vorticity = 1

ns.project_initial_vel = 1
ns.init_pressures      = 1
ns.num_init_passes     = 1
ns.tags_grow           = 1

#initial grids
ns.specifyInitialGrids  = 0
ns.initVelFromVorticity = 1

ns.viscosity =  0.000001

ns.num_scalars = 1
ns.scal_diffusion_coeffs = 0.00

projection.eta = 0.9

# this is physical BC info
# 0 = solidWall, 1=inflow, 2=outflow, 3=symmetry, 4=noShear
physBC.lo = 4 4 4
physBC.hi = 4 4 4
physBC.maxInflowVel = 1.0
```


Figure 1: Inputs file for $32 \times 32 \times 48$ benchmark problem.

obtain the wall-clock time. This method is robust and has a resolution of approximately one microsecond on the target machine.

Another interesting metric is the number of floating-point operations executed per unit of time for a given section of code. This metric is often referred to as the number of MFLOPS (million floating-point operations per second). A common tool used to measure hardware events such as floating-point instructions is PAPI. Currently, the target platform does not have PAPI installed or other tools to measure hardware events. To approximate a floating-point operation rate, we can first obtain the total number of floating-point instructions issued on another platform with PAPI installed and assume that this will be similar to the actual number obtained on the target machine. These values will be different because of compiler and instruction set differences. With the number of floating-point operations for a given section of code, the wall-clock time can be measured on the target machine, and an approximate MFLOP value can be computed. The wall-clock time will be slightly greater than the actual cycle time used, but again, without hardware counters to measure the number of cycles, this is an approximation. The machine used to count the number of floating-point instructions issued was a Pentium III chip with a clock rate of 900 MHZ. The GNU compilers were used on this platform.

Uniprocessor Performance

The driver for the Incompressible Navier Stokes code resides in main.cpp and controls setting up the AMR problem and calling the function that begins the computational solution of the problem. The following table shows timing measurements of these two sections of code for the benchmark problem. We use a boldface font to indicate a label for a section of code that is timed.

Code Section	wall-clock seconds	percent of total	average MFLOPS
Setup AMR	214	4.1	91
Run AMR	4822	94.2	107
totals	5120	98.3	

Table 1: Profile of Top Level Driver

The **Setup AMR** section is executed once and cost of this section overhead is roughly equivalent to one coarse time step. Measurements of problems that use a large number of coarse time steps will show that this overhead is insignificant compared to the the total execution time. Therefore, the total run time can be approximated as that measured in the **Run AMR** section.

Over the entire execution, **Run AMR** can be broken into three distinct sections: **Level Advance**, **Synchronization**, and **Regrid**. The following tables shows the profile of these three sections of code that comprise **Run AMR** for the benchmark problem:

Code Section	wall-clock seconds	percent of parent	average MFLOPS
Level Advance	3135	65.0	108
Synchronization	1318	27.3	101
Regrid	365	7.6	111
totals	4818	99.9	

Table 2: Profile of **AMR Run**

In general, the timing profile for an AMR problem will largely depend on the problem and the input parameters. For example, using Chombo with an input that requests more regridding steps will certainly generate a timing profile that shows more time expended in the **Regrid** section. The maximum number of AMR levels allowed can also affect the profile. For this particular ring problem, the expense of each time step is very sensitive to the vorticity tagging factor. The timing results are presented in a series of tables:

Timed Code Sections			percent of **AMR Run**	total seconds
Level Advance			65.0	3135
	Compute Adv Vel		17.8	851
		Trace State	3.4	165
		Level MAC Proj	14.2	683
	Advect Scalars		4.1	199
	Advect Diff Scalars		4.1	198
	Predict Velocities		6.5	315
	Compute Ustar		17.4	839
		Viscous Solve 1	5.8	280
		Viscous Solve 2	5.8	280
	Level Projection		15.1	726
Synchronization			27.3	1318
	Implicit Reflux		9.2	445
	Composite Proj		9.1	438
	Freestream Corr		8.3	400
Regrid			7.6	365
	Init Velocity Proj		1.3	65
	Init Global Pressure		4.8	231
	Post Regrid Ops		1.1	55

Table 3: Profile of **AMR Run**

The peak memory used for the 32x32x48 problem was about 380MB for a serial run. The 64x64x96 problem used as much as 2080MB of memory in serial.

Timed Code Sections		Pseudo-Code Step
Level Advance		section 2.7.2
	Compute Adv Vel	step 1
	Trace State	step 1a
	Level MAC Proj	steps 1b,1c
	Advect Scalars	step 2
	Advect Diff Scalars	step 2
	Predict Velocities	step 3a
	Compute Ustar	steps 3b,3c,3d
	Viscous Solve 1	step 3c
	Viscous Solve 2	step 3d
	Level Projection	step 4
Synchronization		Section 2.7.2, step 6
	Implicit Reflux	step 6a
	Composite Proj	step 6b
	Freestream Corr	step 6c
Regrid		section 2.7.3
	Init Velocity Proj	step 1
	Init Global Pressure	step 2
	Post Regrid Ops	step 3

Table 4: Matching Timed Sections with Pseudo-Code in Software Design Document

Code Section	wall-clock seconds	percent of total	average MFLOPS
Viscous Solve	404	7.9	108
Sync Projection Solve	412	8.0	106
Freestream Correction Solve	447	8.7	113
Init Velocity Projection Solve	83	1.6	119
Init Data Solve	96	1.7	101
totals	1432	28.0	

Table 5: Diagnostic Multi-Level AMR Solves

The results for the regular-grid operations are an attempt to measure all of the work done in inner regular-grid loops. However, to directly measure these sections of code required calling the timing functions a very large number of times. The timing functions themselves have a small expense associated with them and therefore total execution time for the code was increased when these diagnostic measurements were in place. The total wall-clock time for **AMR Run** with these fine-grained diagnostic measurements was

Code Section	wall-clock seconds	percent of total	average MFLOPS
Init Viscous Solves	59	1.2	134
First Viscous Solve	280	5.5	139
Second Viscous Solve	280	5.5	139
MAC Projection Solve	668	13.0	140
Projection Solve	728	14.2	140
totals	2014	49.3	

Table 6: Diagnostic Single-Level AMR Solves

Code Section	N hits	wall-clock seconds	mod wall-clock seconds	percent of tot	avg. MFLOPS
GSRBLEVLLAP	300848	491	491	10.2	360
GSRBLEVELHELM	2884992	165	163	3.4	397
other Fortran funcs	6245340	344	340	7.1	347
FArrayBox funcs	250428952	908	743	15.4	37
totals	259860132	1908	1736	36.1	203

Table 7: Regular-Grid operations (Fortran routines)

increased by 343 seconds for the benchmark problem. In an effort to account for this inaccuracy, we have modified the wall-clock time spent in each of the timed code sections or group of timed sections. First we assume that the increase in wall-clock time is due solely to the expense of large numbers of timing measurements. Then we assume that half of the expense of a single timing measurement is counted toward the section of code under measurement. Using these assumptions, we have modified the measured wall-clock time of the timed sections by subtracting the following amount:

$$\Delta_{sub} = \frac{\Delta_{diff}}{2} \frac{N_{section}}{N_{total}}, \tag{2}$$

where in this case $\Delta_{diff} = 343$ seconds and $N_{total} = 259860132$.

By way of calibration, we obtained and compiled two common LINPACK benchmarks on several platforms. The LINPACK source was not modified for either size of problem. The compiler used on the Compaq machine is the native Fortran compiler with the `-fast` option. For all other platforms, the GNU Fortran compiler was used with flags `-O3 -fomit-frame-pointer -funroll-loops`.

Machine	N=100 Compiled MFLOPS	N=1000 Compiled MFLOPS	PEAK MFLOPS
Compaq ES45	703	656	2000
Intel P4 2200 MHZ	698	249	4400
AMD Athlon Thunderbird 1.4GHZ	650	92	2800
Intel P3 933 MHZ	345	55	933

Table 8: Compiled LINPACK Results for Machines of Interest

Parallel Performance

To measure the parallel performance of the code, the benchmark problem was run using 1, 4, 16, and 32 processors. The rate column in the table is the total number of points or cells updated in the entire problem divided by the number of processor-seconds, which is simply the wall-clock time multiplied by the number of processors. We also include results for a larger problem of size 64x64x96 with a factor of two decrease in the vorticity tagging factor. The rate would be constant for all problem sizes and number of processors if the parallel scaling of the code was perfect.

Prob size	Vort Tagging Factor	N Points Updated	N Procs	**AMR Run** time (sec)	rate (points/ sec-proc)	unscaled speedup
32x32x48	0.0050	35864576	1	4833	7421	
32x32x48	0.0050	35864576	4	1360	6593	3.6
32x32x48	0.0050	35864576	16	571	3926	8.5
32x32x48	0.0050	35864576	32	403	2781	12.0
64x64x96	0.0025	190840832	1	39638	4815	
64x64x96	0.0025	190840832	16	3019	3951	13.1
64x64x96	0.0025	190840832	32	2699	2210	14.7
64x64x96	0.0025	190840832	64	1988	1500	26.2

Table 9: Parallel Scaling